Numpy Tutorial

1. NumPy in Python

- **NumPy** is a Python library used for working with arrays.
- It also has functions for working in domain of linear algebra, fourier transform, and matrices.
- **NumPy** was created in 2005 by **Travis Oliphant**. It is an **open source** project and you can use it freely.
- **NumPy** can be used to perform a wide variety of **mathematical operations on arrays**.
- It adds powerful data structures to Python that guarantee efficient calculations with arrays and matrices and it supplies **an enormous library of high-level mathematical functions** that operate on these arrays and matrices.
- Besides its obvious scientific uses, **NumPy** can also be used as an efficient multi-dimensional container of generic data.
- **Arbitrary data-types** can be defined using **Numpy** which allows **NumPy** to seamlessly and speedily integrate with a wide variety of databases.
- The **import numpy** portion of the code tells Python to bring the **NumPy** library into your current environment.
- The as np portion of the code then tells Python to give **NumPy** the alias of **np**.
- This allows you to use **NumPy** functions by simply typing np.
- **NumPy** is a very popular python library for large multi-dimensional array and matrix processing, with the help of a large collection of high-level mathematical functions.
- **It is very useful for fundamental scientific computations in Machine Learning.**
- **NumPy** is a **module** for Python.
- The name is an acronym for "Numeric Python" or "Numerical Python".
- It is pronounced /ˈnʌmpaɪ/ (NUM-py) or less often /ˈnʌmpi (NUM-pee)). It
- is an extension module for Python, mostly written in C.
- A **numpy** array is a grid of values, all of the same type, and is indexed by a tuple of nonnegative integers.
- The number of dimensions is the rank of the array; the shape of an array is a tuple of integers giving the size of the array along each dimension.
- **NumPy** is a general-purpose library for working with large arrays and matrices.
- **Scrapy is the most popular high-level Python framework** for extracting data from websites.
- **Matplotlib** is a standard data visualization library that together with **NumPy**, **SciPy**, and **IPython** provides features similar to **MATLAB**.
- **NumPy** (short for **Numerical Python**) provides an efficient interface to store and operate on dense data buffers.
- In some ways, **NumPy** arrays are like Python's built-in list type, but **NumPy** arrays provide much more efficient storage and data operations as the arrays grow larger in size.

NumPy Features

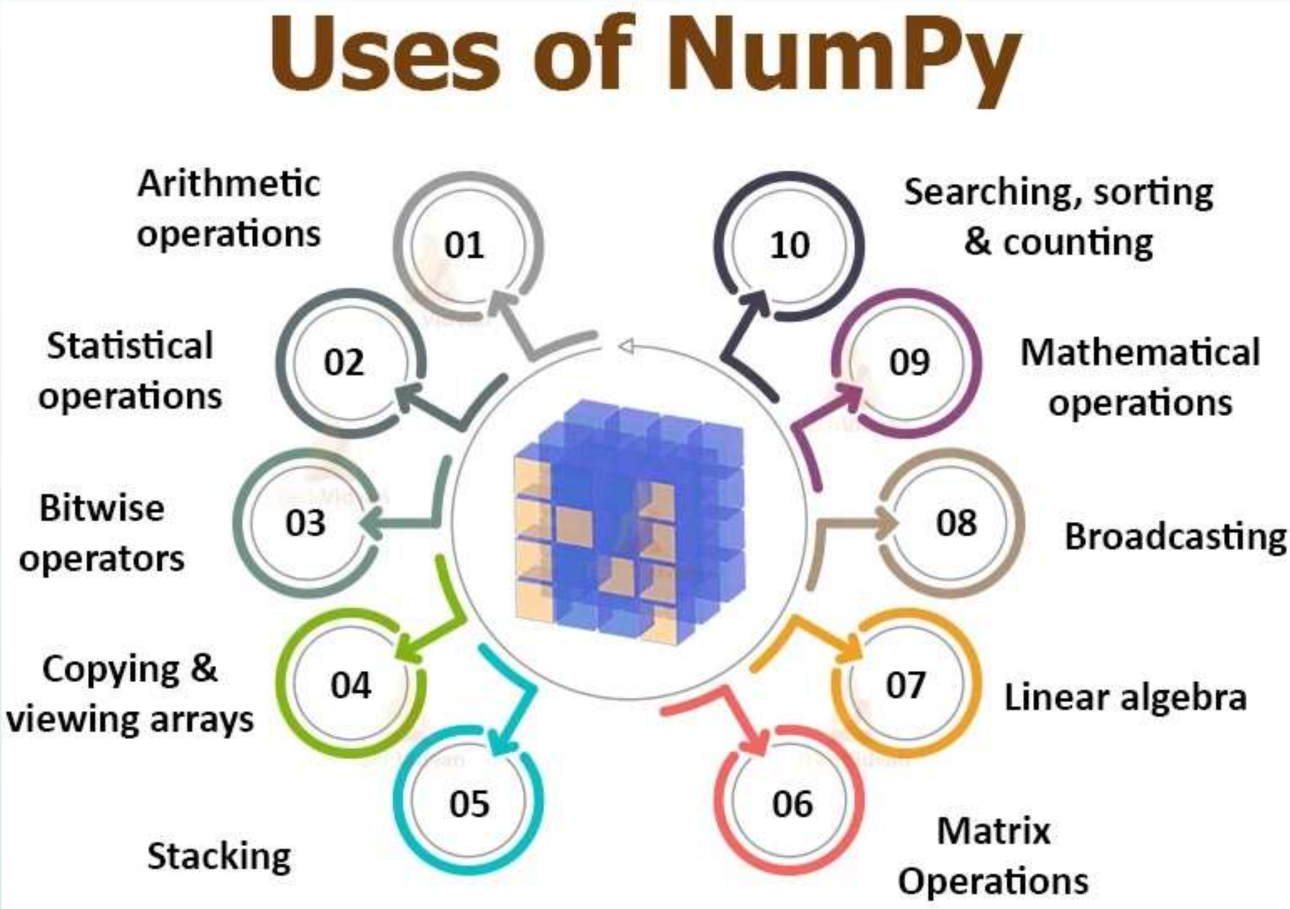
- High-performance N-dimensional array object
- It contains tools for integrating code from C/C++ and Fortran
- It contains a multidimensional container for generic data
- Additional linear algebra, Fourier transform, and random number capabilities
- It consists of broadcasting functions
- It had data type definition capability to work with varied databases

NumPy Arrays are faster than Python Lists because of the following reasons:

- An array is a collection of homogeneous data-types that are stored in contiguous memory locations.
- On the other hand, a list in Python is a collection of heterogeneous data types stored in non-contiguous memory locations.

ndarray attributes

- Every array in NumPy is an object of ndarray class.
- The Properties of an array can be manipulated by accessing the ndarray attributes.
- The more important attributes of an ndarray are ndarray.ndim, ndarray.shape, ndarray.size, ndarray.dtype, and ndarray.itemsize



NumPy data types

- i - integer
- b- boolean
- u - unsigned integer
- f - float
- c - complex float
- m - timedelta
- M - datetime
- O - object
- S - string
- U - Unicode string
- V - fixed chunk of memory for other type

NumPy dtypes

Basic Type	Available NumPy types	Comments
Boolean	bool	Elements are 1 byte in size
Integer	int8, int16, int32, int64, int128, int	int defaults to the size of int in C for the platform
Unsigned Integer	uint8, uint16, uint32, uint64, uint128, uint	uint defaults to the size of unsigned int in C for the platform
Float	float32, float64, float, longfloat,	Float is always a double precision floating point value (64 bits). longfloat represents large precision floats. Its size is platform dependent.
Complex	complex64, complex128, complex	The real and complex elements of a complex64 are each represented by a single precision (32 bit) value for a total size of 64 bits.
Strings	str, unicode	Unicode is always UTF32 (UCS4)
Object	object	Represent items in array as Python objects.
Records	void	Used for arbitrary data structures in record arrays.

import numpy library

In [1]:

```
import numpy as np
```

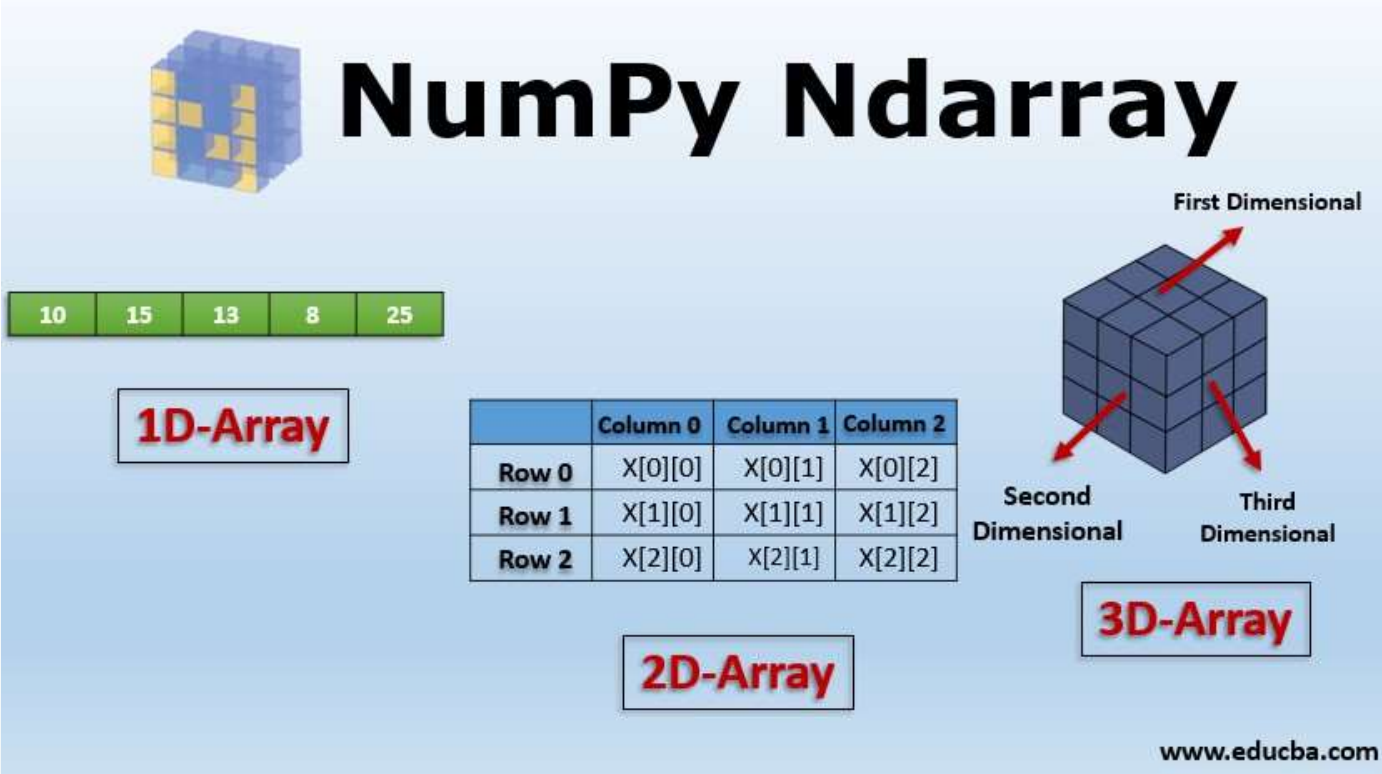
You can find more information about NumPy when executing the following commands.

In [2]:

```
# help(np)
```

Some NumPy methods

- array()
- arange()
- zeros()
- ones()
- empty()
- linspace()



array()

array(data_type, value_list)

In [3]:

```
numpy_array = np.array([0.577, 1.618, 2.718, 3.14, 6, 28, 37, 1729])
print(numpy_array)
print(numpy_array.dtype)
```

```
[5.770e-01 1.618e+00 2.718e+00 3.140e+00 6.000e+00 2.800e+01 3.700e+01
 1.729e+03]
float64
```

In [4]:

```
numpy_array = np.array([0.577, 1.618, 2.718, 3.14, 6, 28, 37, 1729]).reshape(2, 4)
print(numpy_array)
print(numpy_array.dtype)
print()
numpy_array = np.array([0.577, 1.618, 2.718, 3.14, 6, 28, 37, 1729]).reshape(4, 2)
print(numpy_array)
print(numpy_array.dtype)

```

```
[[5.770e-01 1.618e+00 2.718e+00 3.140e+00]
 [6.000e+00 2.800e+01 3.700e+01 1.729e+03]]
float64

[[5.770e-01   1.618e+00]
 [2.718e+00   3.140e+00]
 [6.000e+00   2.800e+01]
 [3.700e+01 1.729e+03]]
float64
```

The following code gives a TypeError.

In [5]:

```
numpy_array = np.array(0.577, 1.618, 2.718, 3.14, 6, 28, 37, 1729)
# This error is due to calling array() mith multiple arguments, instead of a single list of values
print(numpy_array)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_11868/2649635874.py in <module>
----> 1 numpy_array = np.array(0.577, 1.618, 2.718, 3.14, 6, 28, 37, 1729) # This error is due to calling a
rray() mith multiple arguments, instead of a single list of values
      2 print(numpy_array)

TypeError: array() takes from 1 to 2 positional arguments but 8 were given
```

In [ ]:

```
# The following code returns a sequence of two dimensional array
numpy_array = np.array([(0, 1, 1), (2, 3, 5), (8, 13, 21)])
print(numpy_array)
```

```
[[ 0  1  1]
 [ 2  3  5]
 [ 8 13 21]]
```

In [ ]:

```
# To transfrom data type into complex number
numpy_array = np.array([(0, 1, 1), (2, 3, 5), (8, 13, 21)], dtype = complex)
print(numpy_array)
```

```
[[ 0.+0.j  1.+0.j  1.+0.j]
 [ 2.+0.j  3.+0.j  5.+0.j]
 [ 8.+0.j 13.+0.j 21.+0.j]]
```

arange()

np.arange(start, end, step, dtype)

In [ ]:

```
numpy_array = np.arange(0, 100, 5, int)
print(numpy_array)
print(len(numpy_array))
```

```
[ 0  5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90 95]
20
```

In [6]:

```
numpy_array = np.arange(0, 100, 5, int)
numpy_array.shape=(4, 5)
print(numpy_array)
```

```
[[ 0  5 10 15 20]
 [25 30 35 40 45]
 [50 55 60 65 70]
 [75 80 85 90 95]]
```

In [7]:

```
numpy_array = np.arange(0, 100, 5, int).reshape(4, 5)
print(numpy_array)
```

```
[[ 0  5 10 15 20]
 [25 30 35 40 45]
 [50 55 60 65 70]
 [75 80 85 90 95]]
```

In [8]:

```
# Since one of parameters in the array is float type, the elements in the array are of type float.
numpy_array = np.arange(0, 3.14, 0.3)
print(numpy_array)
```

```
[0.  0.3 0.6 0.9 1.2 1.5 1.8 2.1 2.4 2.7 3. ]
```

In [9]:

```
# Type of arange
numpy_array = np.arange(2, 37, 3)
print(numpy_array)
print(type(numpy_array))
```

```
[ 2  5  8 11 14 17 20 23 26 29 32 35]
<class 'numpy.ndarray'>
```

zeros()

np.zeros((shape of the array))

In [10]:

```
numpy_zeros = np.zeros((3, 4))
print(numpy_zeros)
print(type(numpy_zeros))
```

```
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
<class 'numpy.ndarray'>
```

In [11]:

```
numpy_zeros = np.zeros((3, 4), dtype = int)
print(numpy_zeros)
print(type(numpy_zeros))
```

```
[[0 0 0 0]
 [0 0 0 0]
 [0 0 0 0]]
<class 'numpy.ndarray'>
```

In [12]:

```
numpy_zeros = np.zeros((3, 4), dtype = complex)
print(numpy_zeros)
print(type(numpy_zeros))
```

```
[[0.+0.j 0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 0.+0.j 0.+0.j]]
<class 'numpy.ndarray'>
```

In [13]:

```
numpy_zeros = np.zeros((3, 4), dtype = bool)
print(numpy_zeros)
print(type(numpy_zeros))
```

```
[[False False False False]
 [False False False False]
 [False False False False]]
<class 'numpy.ndarray'>
```

ones()

np.ones((shape of the array))

In [14]:

```
numpy_ones = np.ones((3, 4))
print(numpy_ones)
print(type(numpy_ones))
```

```
[[1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]]
<class 'numpy.ndarray'>
```

In [15]:

```
numpy_ones = np.ones((3, 4), dtype =int)
print(numpy_ones)
print(type(numpy_ones))
```

```
[[1 1 1 1]
 [1 1 1 1]
 [1 1 1 1]]
<class 'numpy.ndarray'>
```

In [16]:

```
numpy_ones = np.ones((3, 4), dtype =complex)
print(numpy_ones)
print(type(numpy_ones))
```

```
[[1.+0.j 1.+0.j 1.+0.j 1.+0.j]
 [1.+0.j 1.+0.j 1.+0.j 1.+0.j]
 [1.+0.j 1.+0.j 1.+0.j 1.+0.j]]
<class 'numpy.ndarray'>
```

In [17]:

```
numpy_ones = np.ones((3, 4), dtype =bool)
print(numpy_ones)
print(type(numpy_ones))
```

```
[[ True  True  True  True]
 [ True  True  True  True]
 [ True  True  True  True]]
<class 'numpy.ndarray'>
```

empty()

np.empty((shape of the array))

It creates an empty array in the certain dimension with random values which change for every call.

In [18]:

```
numpy_empty = np.empty((4, 4))
print(numpy_empty)
```

```
[[4.67296746e-307 1.69121096e-306 1.69119330e-306 1.42413555e-306]
 [1.78019082e-306 1.37959740e-306 6.23057349e-307 1.02360935e-306]
 [1.69120416e-306 1.78022342e-306 6.23058028e-307 1.06811422e-306]
 [9.45699680e-308 1.11261027e-306 1.37961913e-306 9.34604358e-307]]
```

In [19]:

```
numpy_empty = np.empty((4, 4), dtype = int)
print(numpy_empty)
```

```
[[-1271310320  1071806152  -137438953  1073341267]
 [ -927712936  1074118262  1374389535  1074339512]
 [          0  1075314688           0  1077673984]
 [          0  1078099968           0  1083900928]]
```

In [20]:

```
numpy_empty = np.empty((4, 4), dtype = complex)
print(numpy_empty)
```

```
[[1.23160026e-311+1.05730048e-321j 0.00000000e+000+0.00000000e+000j
  1.42413554e-306+5.02034658e+175j 1.21004824e-071+4.23257854e+175j]
 [3.41996727e-032+4.90863814e-062j 9.83255598e-072+4.25941885e-096j
  1.12855837e+277+8.93168725e+271j 7.33723594e+223+1.70098498e+256j]
 [5.49109388e-143+1.06396443e+224j 3.96041428e+246+1.16318408e-028j
  1.89935647e-052+9.85513351e+165j 1.08805205e-071+4.18109207e-062j]
 [2.24151504e+174+3.36163259e-067j 5.41760579e-067+3.18070066e-028j
  3.93896263e-062+5.74015544e+180j 1.94919988e-153+1.02847381e-307j]]
```

In [21]:

```
numpy_empty = np.empty((4, 4), dtype = bool)
print(numpy_empty)
```

```
[[ True False  True  True]
 [ True  True  True  True]
 [ True  True  True  True]
 [ True  True  True  True]]
```

linspace()

np.linspace(start, stop, num, endpoint, retstep, dtype, axis)

It creates an array with evenly spaced values within specified interval.

In [22]:

```
numpy_linspace = np.linspace(1, 37, 37)
print(numpy_linspace)
```

```
[ 1.  2.  3.  4.  5.  6.  7.  8.  9. 10. 11. 12. 13. 14. 15. 16. 17. 18.
 19. 20. 21. 22. 23. 24. 25. 26. 27. 28. 29. 30. 31. 32. 33. 34. 35. 36.
 37.]
```

In [23]:

```
numpy_linspace = np.linspace(1, 37, 37, dtype=int)
print(numpy_linspace)
```

```
[ 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
 25 26 27 28 29 30 31 32 33 34 35 36 37]
```

In [24]:

```
numpy_linspace = np.linspace(1, 37, 37, dtype=bool)
print(numpy_linspace)
```

```
[ True  True  True  True  True  True  True  True  True  True  True  True
  True  True  True  True  True  True  True  True  True  True  True  True
  True  True  True  True  True  True  True  True  True  True  True  True
  True]
```

In [25]:

```
numpy_linspace = np.linspace(1, 37, 37, dtype=complex)
print(numpy_linspace)
```

```
[ 1.+0.j  2.+0.j  3.+0.j  4.+0.j  5.+0.j  6.+0.j  7.+0.j  8.+0.j  9.+0.j
 10.+0.j 11.+0.j 12.+0.j 13.+0.j 14.+0.j 15.+0.j 16.+0.j 17.+0.j 18.+0.j
 19.+0.j 20.+0.j 21.+0.j 22.+0.j 23.+0.j 24.+0.j 25.+0.j 26.+0.j 27.+0.j
 28.+0.j 29.+0.j 30.+0.j 31.+0.j 32.+0.j 33.+0.j 34.+0.j 35.+0.j 36.+0.j
 37.+0.j]
```

reshape()

np.reshape(line_number, column_number, order = 'C')

In [26]:

```
numpy_arange = np.arange(1, 37).reshape(6, 6)
print(numpy_arange)
```

```
[[ 1  2  3  4  5  6]
 [ 7  8  9 10 11 12]
 [13 14 15 16 17 18]
 [19 20 21 22 23 24]
 [25 26 27 28 29 30]
 [31 32 33 34 35 36]]
```

In [50]:

```
numpy_exercises = np.arange(16).reshape(4, 4)
print(numpy_exercises)
print(numpy_exercises.size)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]
16
```

In [28]:

```
"""
If the array size is very large then only corners of the array are printed.
Then the central part of the array is skipped.
"""
numpy_exercises = np.arange(10000).reshape(200, 50)
print(numpy_exercises)
```

```
[[   0    1    2 ...   47   48   49]
 [  50   51   52 ...   97   98   99]
 [ 100  101  102 ...  147  148  149]
 ...
 [9850 9851 9852 ... 9897 9898 9899]
 [9900 9901 9902 ... 9947 9948 9949]
 [9950 9951 9952 ... 9997 9998 9999]]
```

set_printoption()

np.set_printoption(threshold = sys.maxsize)

In [29]:

```
"""
To enable numpy to print the entire array, use 'set_printoptions()' method.
"""
import sys
numpy_exercises = np.arange(10000).reshape(200, 50)
np.set_printoptions(threshold = sys.maxsize)
print(numpy_exercises)
```

```
[[  0   1   2   3   4   5   6   7   8   9  10  11  12  13
   14  15  16  17  18  19  20  21  22  23  24  25  26  27
   28  29  30  31  32  33  34  35  36  37  38  39  40  41
   42  43  44  45  46  47  48  49]
 [ 50  51  52  53  54  55  56  57  58  59  60  61  62  63
   64  65  66  67  68  69  70  71  72  73  74  75  76  77
   78  79  80  81  82  83  84  85  86  87  88  89  90  91
   92  93  94  95  96  97  98  99]
 [100 101 102 103 104 105 106 107 108 109 110 111 112 113
  114 115 116 117 118 119 120 121 122 123 124 125 126 127
  128 129 130 131 132 133 134 135 136 137 138 139 140 141
  142 143 144 145 146 147 148 149]
 [150 151 152 153 154 155 156 157 158 159 160 161 162 163
  164 165 166 167 168 169 170 171 172 173 174 175 176 177
  178 179 180 181 182 183 184 185 186 187 188 189 190 191
  192 193 194 195 196 197 198 199]
 [200 201 202 203 204 205 206 207 208 209 210 211 212 213
  214 215 216 217 218 219 220 221 222 223 224 225 226 227
  228 229 230 231 232 233 234 235 236 237 238 239 240 241
```

indexing

Indexing in Python starts with 0.

In [30]:

```
# Using 1D array
special_nums = np.array([0.577, 1.618, 2.718, 3.14, 6, 37, 1729])
print(special_nums[0])
print(special_nums[-1])     # This shows negative indexing and negative indexing starts with -1.
print(special_nums[-3])
print(special_nums[2])
print(special_nums[5])
```

```
0.577
1729.0
6.0
2.718
37.0
```

In [31]:

```
# Using 2D array
special_nums = np.array([0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]).reshape(3, 4)
print(special_nums)
print(len(special_nums))
print(special_nums[1, 1])      # It means first line, first column, namely the output is 5.
print(special_nums[2, 3])      # It means second line, third column, namely the output is 89.
```

```
[[ 0  1  1  2]
 [ 3  5  8 13]
 [21 34 55 89]]
3
5
89
```

Addition

In [32]:

```
numpy_arange = np.arange(20, 50, 7.5).reshape(2, 2)
numpy_array = np.array([0.577, 1.618, 2.718, 3.14]).reshape(2, 2)
print(numpy_arange)
print()
print(numpy_array)
print()
print('Added array')
addition = numpy_arange+numpy_array    # Addition
print(addition)
```

```
[[20.  27.5]
 [35.  42.5]]

[[0.577 1.618]
 [2.718 3.14 ]]

Added array
[[20.577 29.118]
 [37.718 45.64 ]]
```

Substraction

In [33]:

```
numpy_arange = np.arange(20, 50, 7.5).reshape(2, 2)
numpy_array = np.array([0.577, 1.618, 2.718, 3.14]).reshape(2, 2)
print(numpy_arange)
print()
print(numpy_array)
print()
print('Substracted array')
substraction = numpy_arange-numpy_array    # Substraction
print(substraction)
```

```
[[20.  27.5]
 [35.  42.5]]

[[0.577 1.618]
 [2.718 3.14 ]]

Substracted array
[[19.423 25.882]
 [32.282 39.36 ]]
```

Multiplication

In [34]:

```
numpy_arange = np.arange(20, 50, 7.5).reshape(2, 2)
numpy_array = np.array([0.577, 1.618, 2.718, 3.14]).reshape(2, 2)
print(numpy_arange)
print()
print(numpy_array)
print()
print('Multiplicated array')
multiplication = numpy_arange*numpy_array    # Multiplication
print(multiplication)
```

```
[[20.  27.5]
 [35.  42.5]]

[[0.577 1.618]
 [2.718 3.14 ]]

Multiplicated array
[[ 11.54   44.495]
 [ 95.13  133.45 ]]
```

Division

In [35]:

```
numpy_arange = np.arange(20, 50, 7.5).reshape(2, 2)
numpy_array = np.array([0.577, 1.618, 2.718, 3.14]).reshape(2, 2)
print(numpy_arange)
print()
print(numpy_array)
print()
print('Divided array')
division = numpy_arange/numpy_array   # Substraction
print(division)
```

```
[[20.  27.5]
 [35.  42.5]]

[[0.577 1.618]
 [2.718 3.14 ]]

Divided array
[[34.66204506 16.99629172]
 [12.87711553 13.53503185]]
```

Floor division

In [36]:

```
numpy_arange = np.arange(20, 50, 7.5).reshape(2, 2)
numpy_array = np.array([0.577, 1.618, 2.718, 3.14]).reshape(2, 2)
print(numpy_arange)
print()
print(numpy_array)
print()
print('Divisor array')
divisor = numpy_arange//numpy_array   # Floor division
print(divisor)
```

```
[[20.  27.5]
 [35.  42.5]]

[[0.577 1.618]
 [2.718 3.14 ]]

Divisor array
[[34. 16.]
 [12. 13.]]
```

Modulus

In [37]:

```
numpy_arange = np.arange(20, 50, 7.5).reshape(2, 2)
numpy_array = np.array([0.577, 1.618, 2.718, 3.14]).reshape(2, 2)
print(numpy_arange)
print()
print(numpy_array)
print()
print('Modulus array')
modulus = numpy_arange%numpy_array   # Modulus
print(modulus)
```

```
[[20.  27.5]
 [35.  42.5]]

[[0.577 1.618]
 [2.718 3.14 ]]

Modulus array
[[0.382 1.612]
 [2.384 1.68 ]]
```

Exponentiation

In [38]:

```
numpy_arange = np.arange(20, 50, 7.5).reshape(2, 2)
numpy_array = np.array([0.577, 1.618, 2.718, 3.14]).reshape(2, 2)
print(numpy_arange)
print()
print(numpy_array)
print()
print('Exponentiated array')
exponentiation = numpy_arange**numpy_array   # Exponentiation
print(exponentiation)
```

```
[[20.  27.5]
 [35.  42.5]]

[[0.577 1.618]
 [2.718 3.14 ]]

Exponentiated array
[[5.63241060e+00 2.13226068e+02]
 [1.57317467e+04 1.29760121e+05]]
```

@ operator

new_matrix = matrix_1 * matrix_2

Returns the product of the two matrices.

In [39]:

```
numpy_arange = np.arange(20, 50, 7.5).reshape(2, 2)
numpy_array = np.array([0.577, 1.618, 2.718, 3.14]).reshape(2, 2)
print(numpy_arange)
print()
print(numpy_array)
print()
print('Product of @ operator')
array = numpy_arange@numpy_array   # @ operator
print(array)
```

```
[[20.  27.5]
 [35.  42.5]]

[[0.577 1.618]
 [2.718 3.14 ]]

Product of @ operator
[[ 86.285 118.71 ]
 [135.71  190.08 ]]
```

dot()

new_matrix = matrix_1.dot(matrix_2)

It returns the product of the two matrices. It is as same as the @ operator.

In [40]:

```
numpy_arange = np.arange(20, 50, 7.5).reshape(2, 2)
numpy_array = np.array([0.577, 1.618, 2.718, 3.14]).reshape(2, 2)
print(numpy_arange)
print()
print(numpy_array)
print()
print('Product of the function dot()')
array = numpy_arange.dot(numpy_array)   # dot() function
print(array)
```

```
[[20.  27.5]
 [35.  42.5]]

[[0.577 1.618]
 [2.718 3.14 ]]

Product of the function dot()
[[ 86.285 118.71 ]
 [135.71  190.08 ]]
```

Relational operations

array_name, operator, value

numpy_array<100

In [49]:

```
numpy_array = np.arange(10, 50, 5)
print(numpy_array < 30)
print(numpy_array > 30)
print(numpy_array == 30)
print(numpy_array != 30)
print(numpy_array >= 30)
print(numpy_array <= 30)
```

```
[ True  True  True  True False False False False]
[False False False False False  True  True  True]
[False False False False  True False False False]
[ True  True  True  True False  True  True  True]
[False False False False  True  True  True  True]
[ True  True  True  True  True False False False]
```

Operations with different types of arrays

It returns a UFuncTypeError.

In [56]:

```
numpy_array_one = np.ones((2, 2), dtype=int)
numpy_array_two = np.arange(2, 20.0)
numpy_array_one += numpy_array_two
print(numpy_array_one)
```

```
---------------------------------------------------------------------------
UFuncTypeError                            Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_11868/3374378000.py in <module>
      1 numpy_array_one = np.ones((2, 2), dtype=int)
      2 numpy_array_two = np.arange(2, 20.0)
----> 3 numpy_array_one += numpy_array_two
      4 print(numpy_array_one)

UFuncTypeError: Cannot cast ufunc 'add' output from dtype('float64') to dtype('int32') with casting rule 'same_kind'
```

Unary operations

- array_name.min()
- array_name.max()
- array_name.sum()
- etc.

In [92]:

```python
numpy_array = np.arange(1, 25).reshape(4, 6)
print(numpy_array)
print(f'The minimum element of the certain array is {numpy_array.min()}.')
print(f'The maximum element of the certain array is {numpy_array.max()}.')
print(f'The sum of the elements of the array is {numpy_array.sum()}.')
print(f'The mean of the elements of the array is {numpy_array.mean()}.')
print(f'The standard deviation of the elements of the array is {numpy_array.std()}.')
print(f'The variance of the elements of the array is {numpy_array.var()}.')
print(f'The length of the elements of the array is {len(numpy_array)}.')
print(f'The shape of the array is {numpy_array.shape}.')
print(f'The dtype of the array is {numpy_array.dtype}.')
print(f'The type of the array is {type(numpy_array)}.')
print(f'The minimum numbers of every row are {numpy_array.min(axis = 1)}.')     # axis = 1 denotes the row.
print(f'The maximum numbers of every row are {numpy_array.max(axis = 1)}.')
print(f'The minimum numbers of every column are {numpy_array.min(axis=0)}.')   # axis = 0 denotes the column
print(f'The maximum numbers of every column are {numpy_array.max(axis=0)}.')
print(f'The sum of the numbers in each column are {numpy_array.sum(axis=0)}.')
print(f'The sum of the numbers in each row are {numpy_array.sum(axis=1)}.')
print(f'The mean of the numbers in each column is {numpy_array.mean(axis=0)}.')
print(f'The mean of the numbers in each row is {numpy_array.mean(axis=1)}.')
```

```
[[ 1  2  3  4  5  6]
 [ 7  8  9 10 11 12]
 [13 14 15 16 17 18]
 [19 20 21 22 23 24]]
The minimum element of the certain array is 1.
The maximum element of the certain array is 24.
The sum of the elements of the array is 300.
The mean of the elements of the array is 12.5.
The standard deviation of the elements of the array is 6.922186552431729.
The variance of the elements of the array is 47.916666666666664.
The length of the elements of the array is 4.
The shape of the array is (4, 6).
The dtype of the array is int32.
The type of the array is <class 'numpy.ndarray'>.
The minimum numbers of every row are [ 1  7 13 19].
The maximum numbers of every row are [ 6 12 18 24].
The minimum numbers of every column are [1 2 3 4 5 6].
The maximum numbers of every column are [19 20 21 22 23 24].
The sum of the numbers in each column are [40 44 48 52 56 60].
The sum of the numbers in each row are [ 21  57  93 129].
The mean of the numbers in each column is [10. 11. 12. 13. 14. 15.].
The mean of the numbers in each row is [ 3.5  9.5 15.5 21.5].
```

In [93]:

```
print(f'The cumulative sum of the numbers in each column is \n {numpy_array.cumsum(axis=0)}.')
print(f'The cumulative sum of the numbers in each row is \n {numpy_array.cumsum(axis=1)}.')
print(f'The cumulative sum of the numbers in the array is \n {numpy_array.cumsum()}.')
print('etc...')
```

```
The cumulative sum of the numbers in each column is
 [[ 1  2  3  4  5  6]
 [ 8 10 12 14 16 18]
 [21 24 27 30 33 36]
 [40 44 48 52 56 60]].
The cumulative sum of the numbers in each row is
 [[  1   3   6  10  15  21]
 [  7  15  24  34  45  57]
 [ 13  27  42  58  75  93]
 [ 19  39  60  82 105 129]].
The cumulative sum of the numbers in the array is
 [  1   3   6  10  15  21  28  36  45  55  66  78  91 105 120 136 153 171
 190 210 231 253 276 300].
etc...
```

Assignment operators in array

In [111]:

```
import math
numpy_array = np.arange(1, 25, 3)
print('The main array is', numpy_array)
numpy_array_addition = numpy_array + 6
print('By using the += operator, the array is', numpy_array_addition)
numpy_array_subtraction = numpy_array - 6
print('By using the -= operator, the array is', numpy_array_subtraction)
numpy_array_multiplication = numpy_array * 6
print('By using the *= operator, the array is', numpy_array_multiplication)
numpy_array_division = numpy_array / 6
print('By using the /= operator, the array is', numpy_array_division)
numpy_array_floor_division = numpy_array // 6
print('By using the //= operator, the array is', numpy_array_floor_division)
numpy_array_modulus = numpy_array % 6
print('By using the %= operator, the array is', numpy_array_modulus)
numpy_array_exponentiation = numpy_array ** 6
print('By using the **= operator, the array is', numpy_array_exponentiation)

```

```
The main array is [ 1  4  7 10 13 16 19 22]
By using the += operator, the array is [ 7 10 13 16 19 22 25 28]
By using the -= operator, the array is [-5 -2  1  4  7 10 13 16]
By using the *= operator, the array is [  6  24  42  60  78  96 114 132]
By using the /= operator, the array is [0.16666667 0.66666667 1.16666667 1.66666667 2.16666667 2.66666667
 3.16666667 3.66666667]
By using the //= operator, the array is [0 0 1 1 2 2 3 3]
By using the %= operator, the array is [1 4 1 4 1 4 1 4]
By using the **= operator, the array is [        1      4096    117649   1000000   4826809  16777216  47045881
 113379904]
```

In [125]:

```
numpy_array_one = np.arange(1, 10).reshape(3, 3)
print('The first main array is \n', numpy_array_one)
numpy_array_two = np.arange(11, 20).reshape(3, 3)
print('The second main array is \n', numpy_array_two)
new_array_addition = numpy_array_one + numpy_array_two
print('By using the += operator, the new array will be \n', new_array_addition)
new_array_substraction = numpy_array_one - numpy_array_two
print('By using the -= operator, the new array will be \n', new_array_substraction)
new_array_multiplication = numpy_array_one * numpy_array_two
print('By using the *= operator, the new array will be \n', new_array_multiplication)
```

```
The first main array is 
 [[1 2 3]
 [4 5 6]
 [7 8 9]]
The second main array is 
 [[11 12 13]
 [14 15 16]
 [17 18 19]]
By using the += operator, the new array will be 
 [[12 14 16]
 [18 20 22]
 [24 26 28]]
By using the -= operator, the new array will be 
 [[-10 -10 -10]
 [-10 -10 -10]
 [-10 -10 -10]]
By using the *= operator, the new array will be 
 [[ 11  24  39]
 [ 56  75  96]
 [119 144 171]]
```

In [126]:

```
new_array_division = numpy_array_one / numpy_array_two
print('By using the /= operator, the new array will be \n', new_array_division)
new_array_floor_division = numpy_array_one // numpy_array_two
print('By using the //= operator, the new array will be \n', new_array_floor_division)
new_array_modules = numpy_array_one % numpy_array_two
print('By using the %= operator, the new array will be \n', new_array_modules)
new_array_exponentiation = numpy_array_one ** numpy_array_two
print('By using the **= operator, the new array will be \n', new_array_exponentiation)
```

```
By using the /= operator, the new array will be
 [[0.09090909 0.16666667 0.23076923]
 [0.28571429 0.33333333 0.375     ]
 [0.41176471 0.44444444 0.47368421]]
By using the //= operator, the new array will be
 [[0 0 0]
 [0 0 0]
 [0 0 0]]
By using the %= operator, the new array will be
 [[1 2 3]
 [4 5 6]
 [7 8 9]]
By using the **= operator, the new array will be
 [[          1        4096     1594323]
 [  268435456   452807053  -683606016]
 [-2094633337           0  -400556711]]
```

concatenate()

np.concatenate((array_one, array_two,....), axis=0 or 1)

In [135]:

```
numpy_array_one = np.arange(1, 10).reshape(3, 3)
numpy_array_two = np.arange(11, 20).reshape(3, 3)
new_array_one = np.concatenate((numpy_array_one, numpy_array_two), axis=1)
print('This is the array one obtained using axis = 1 \n', new_array_one)
new_array_two = np.concatenate((numpy_array_one, numpy_array_two), axis=0)
print('This is the array two obtained using axis = 0 \n', new_array_two)
```

```
This is the array one obtained using axis = 1
 [[ 1  2  3 11 12 13]
 [ 4  5  6 14 15 16]
 [ 7  8  9 17 18 19]]
This is the array two obtained using axis = 0
 [[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [11 12 13]
 [14 15 16]
 [17 18 19]]
```

In [140]:

```
x = np.array([[0.577, 1.618], [2.718, 3.14]])
y = np.array([[6, 28], [37, 1729]])
z = np.concatenate((x, y), axis = 1)
print('This is the new array yielded by adding into the row \n', z)
z = np.concatenate((x, y), axis = 0)
print('This is the new array yielded by adding into the column \n', z)
```

```
This is the new array by adding into the row
 [[5.770e-01 1.618e+00 6.000e+00 2.800e+01]
 [2.718e+00 3.140e+00 3.700e+01 1.729e+03]]
This is the new array by adding into the column
 [[5.770e-01 1.618e+00]
 [2.718e+00   3.140e+00]
 [6.000e+00   2.800e+01]
 [3.700e+01 1.729e+03]]
```

Splitting of 1D arrays

np.array_split(array_name, number_of_splits)

In [143]:

```
array_special_nums = np.array([0.577, 1.618, 2, 2.718, 3.14, 6, 28, 37, 1729])
print('Before splitting the array \n', array_special_nums)
new_array = np.array_split(array_special_nums, 3)
print('After splitting the array \n',new_array)
```

```
Before splitting the array
 [5.770e-01 1.618e+00 2.000e+00 2.718e+00 3.140e+00 6.000e+00 2.800e+01
 3.700e+01 1.729e+03]
After splitting the array
 [array([0.577, 1.618, 2.   ]), array([2.718, 3.14 , 6.   ]), array([  28.,   37., 1729.])]
```

In [144]:

```
array_special_nums = np.array([0.577, 1.618, 2, 2.718, 3.14, 6, 28, 37, 1729])
print('Before splitting the array \n', array_special_nums)
new_array = np.array_split(array_special_nums, 6)
print('After splitting the array \n',new_array)
```

```
Before splitting the array
 [5.770e-01 1.618e+00 2.000e+00 2.718e+00 3.140e+00 6.000e+00 2.800e+01
 3.700e+01 1.729e+03]
After splitting the array
 [array([0.577, 1.618]), array([2.   , 2.718]), array([3.14, 6.  ]), array([28.]), array([37.]), array([1729.])]
```

Splitting of 2D arrays

np.array_split(array_name, number_of_splits, axis=0 or 1)

In [146]:

```
array_special_nums = np.array([[0.577, 1.618], [2, 2.718], [3.14, 6], [13, 28], [37, 1729]])
print('Before splitting the array \n', array_special_nums)
new_array = np.array_split(array_special_nums, 2)
print('After splitting the array \n',new_array)
```

```
Before splitting the array
 [[5.770e-01 1.618e+00]
 [2.000e+00 2.718e+00]
 [3.140e+00 6.000e+00]
 [1.300e+01 2.800e+01]
 [3.700e+01 1.729e+03]]
After splitting the array
 [array([[0.577, 1.618],
       [2.   , 2.718],
       [3.14 , 6.   ]]), array([[  13.,   28.],
       [  37., 1729.]])]
```

In [147]:

```
array_special_nums = np.array([[0.577, 1.618], [2, 2.718], [3.14, 6], [13, 28], [37, 1729]])
print('Before splitting the array \n', array_special_nums)
new_array = np.array_split(array_special_nums, 2, axis=0)
print('After splitting the array \n',new_array)
```

```
Before splitting the array
 [[5.770e-01 1.618e+00]
 [2.000e+00 2.718e+00]
 [3.140e+00 6.000e+00]
 [1.300e+01 2.800e+01]
 [3.700e+01 1.729e+03]]
After splitting the array
 [array([[0.577, 1.618],
       [2.   , 2.718],
       [3.14 , 6.   ]]), array([[  13.,   28.],
       [  37., 1729.]])]
```

In [148]:

```
array_special_nums = np.array([[0.577, 1.618], [2, 2.718], [3.14, 6], [13, 28], [37, 1729]])
print('Before splitting the array \n', array_special_nums)
new_array = np.array_split(array_special_nums, 2, axis=1)
print('After splitting the array \n',new_array)
```

```
Before splitting the array
 [[5.770e-01 1.618e+00]
 [2.000e+00 2.718e+00]
 [3.140e+00 6.000e+00]
 [1.300e+01 2.800e+01]
 [3.700e+01 1.729e+03]]
After splitting the array
 [array([[ 0.577],
       [ 2.   ],
       [ 3.14 ],
       [13.   ],
       [37.   ]]), array([[1.618e+00],
       [2.718e+00],
       [6.000e+00],
       [2.800e+01],
       [1.729e+03]])]
```

Indexing to get subarrays

In [2]:

```
import numpy as np
array_special_nums = np.array([0.577, 1.618, 2, 2.718, 3.14, 6, 28, 37, 1729])
splitted_array = np.array_split(array_special_nums, 5)
print('Before indexing\n', splitted_array)
print('After indexing\n', splitted_array[0:2])
print('After splitting\n', splitted_array[2:5])
print('After splitting\n', splitted_array[1])
print('After splitting\n', splitted_array[2])
```

```
Before indexing
 [array([0.577, 1.618]), array([2.   , 2.718]), array([3.14, 6.  ]), array([28., 37.]), array([1729.])]
After indexing
 [array([0.577, 1.618]), array([2.   , 2.718])]
After splitting
 [array([3.14, 6.  ]), array([28., 37.]), array([1729.])]
After splitting
 [2.    2.718]
After splitting
 [3.14 6.  ]
```

Copy of array

- **With assignment:** new_array = old_array
- **With shallow copy:** new_array = old_array.view()
- **With deep copy:** new_array = old_array.copy()

In [17]:

```
# With assignment
array_old = np.array([0, 1, 1, 2, 3, 5, 8, 13, 21, 34])
new_array = array_old
print('The old array is', array_old, 'and the id of the old array is', id(array_old))
print('The new array is', new_array, 'and the id of the new array is', id(new_array), 'which is same as that of the old ar
print(id(array_old))
for i in array_old:
    print(i, end=' ')
print()
print(id(new_array))
for i in new_array:
    print(i, end=' ')
```

```
The old array is [ 0  1  1  2  3  5  8 13 21 34] and the id of the old array is 2017306300016
The new array is [ 0  1  1  2  3  5  8 13 21 34] and the id of the new array is 2017306300016 which is sam
e as that of the old array.
2017306300016
0 1 1 2 3 5 8 13 21 34
2017306300016
0 1 1 2 3 5 8 13 21 34
```

In [20]:

```
# With shallow copy
array_old = np.array([0, 1, 1, 2, 3, 5, 8, 13, 21, 34])
new_array = array_old.view()
print('The old array is', array_old, 'and the id of the old array is', id(array_old))
print('The new array is', new_array, 'and the id of the new array is', id(new_array), 'which is different from that of the
```

```
The old array is [ 0  1  1  2  3  5  8 13 21 34] and the id of the old array is 2017306299056
The new array is [ 0  1  1  2  3  5  8 13 21 34] and the id of the new array is 2017306299248 which is diffe
rent from that of the old array.
```

In [21]:

```
# With deep copy
array_old = np.array([0, 1, 1, 2, 3, 5, 8, 13, 21, 34])
new_array = array_old.copy()
print('The old array is', array_old, 'and the id of the old array is', id(array_old))
print('The new array is', new_array, 'and the id of the new array is', id(new_array), 'which is different from that of the
```

```
The old array is [ 0  1  1  2  3  5  8 13 21 34] and the id of the old array is 2017306297904
The new array is [ 0  1  1  2  3  5  8 13 21 34] and the id of the new array is 2017306296944 which is diffe
rent from that of the old array.
```

Searching with where() method

np.where(array_name==element to search)

In [24]:

```
fibonacci_nums = np.array([0, 1, 1, 2, 3, 5, 8, 13, 21, 34])
new_array_one = np.where(fibonacci_nums==1)
new_array_two = np.where(fibonacci_nums==8)
print(new_array_one)
print(new_array_two)
```

```
(array([1, 2], dtype=int64),)
(array([6], dtype=int64),)
```

In [25]:

```
fibonacci_nums = np.array([0, 1, 1, 2, 3, 5, 8, 13, 21, 34])
new_array_one = np.where(fibonacci_nums %2==1)
new_array_two = np.where(fibonacci_nums %3==0)
print(new_array_one)
print(new_array_two)
```

```
(array([1, 2, 4, 5, 7, 8], dtype=int64),)
(array([0, 4, 8], dtype=int64),)
```

In [26]:

```
fibonacci_nums = np.array([0, 1, 1, 2, 3, 5, 8, 13, 21, 34])
new_array_one = np.where(fibonacci_nums //2==1)
new_array_two = np.where(fibonacci_nums //3==0)
print(new_array_one)
print(new_array_two)
```

```
(array([3, 4], dtype=int64),)
(array([0, 1, 2, 3], dtype=int64),)
```

Searching with searchsorted() method

np.searchsorted(name_array, value)

In [28]:

```
fibonacci_nums = np.array([0, 1, 1, 2, 3, 5, 8, 13, 21, 34])
new_array = np.searchsorted(fibonacci_nums, 2)
print(f' The element 2 in fibonacci numbers is in index {new_array}.')
```

```
 The element 2 in fibonacci numbers is in index 3.
```

Sorting

np.sort(name_array)

In [29]:

```
mix_fibonacci_nums = np.array([21, 34, 0, 2, 5, 3,  8, 1, 1, 13])
print('Before sorting\n', mix_fibonacci_nums)
sorted_fibonacci_nums = np.sort(mix_fibonacci_nums)
print('After sorting\n', sorted_fibonacci_nums)

```

```
Before sorting
 [21 34  0  2  5  3  8  1  1 13]
After sorting
 [ 0  1  1  2  3  5  8 13 21 34]
```

In [44]:

```
fruit_array = np.array(['Banana', 'Orange', 'Erdberry', 'Apple', 'Pineapple', 'Kiwi'])
print('Before sorting\n', fruit_array)
fruit_array = np.sort(fruit_array)
print('After sorting\n', fruit_array)
```

```
Before sorting
 ['Banana' 'Orange' 'Erdberry' 'Apple' 'Pineapple' 'Kiwi']
After sorting
 ['Apple' 'Banana' 'Erdberry' 'Kiwi' 'Orange' 'Pineapple']
```

In [47]:

```
mix_fibonacci_nums_2D_array = np.array([[21, 34, 0], [2, 5, 3], [8, 1,1]])
print('Before sorting\n', mix_fibonacci_nums_2D_array)
mix_fibonacci_nums_2D_array = np.sort(mix_fibonacci_nums_2D_array)
print('After sorting\n', mix_fibonacci_nums_2D_array)

```

```
Before sorting
 [[21 34  0]
 [ 2  5  3]
 [ 8  1  1]]
After sorting
 [[ 0 21 34]
 [ 2  3  5]
 [ 1  1  8]]
```

Statistics

- np.mean(array)
- np.max(array)
- np.min(array)
- np.sum(array)
- np.std(array)
- np.var(array)
- np.median(array)

In [52]:

```
fibonacci_nums = np.array([0, 1, 1, 2, 3, 5, 8, 13, 21, 34])
mean = np.mean(fibonacci_nums)
print(f'The mean is {mean}.')
maximum = np.max(fibonacci_nums)
print(f'The maximum is {maximum}.')
minimum = np.min(fibonacci_nums)
print(f'The minimum is {minimum}.')
total = np.sum(fibonacci_nums)
print(f'The sum is {total}.')
standard_deviation = np.std(fibonacci_nums)
print(f'The standard deviation is {standard_deviation}.')
variance = np.var(fibonacci_nums)
print(f'The variance is {variance}.')
median = np.median(fibonacci_nums)
print(f'The median is {median}.')
print(f'The size of the array is {fibonacci_nums.size}.')
print(f'The length of the array is {len(fibonacci_nums)}.')
```

```
The mean is 8.8.
The maximum is 34.
The minimum is 0.
The sum is 88.
The standard deviation is 10.467091286503619.
The variance is 109.55999999999999.
The median is 4.0.
The size of the array is 10.
The length of the array is 10.
```

Mathematical functions

In [59]:

```
print(np.pi)
print(np.e)
print(np.nan)
print(np.inf)
print(-np.inf)
```

```
3.141592653589793
2.718281828459045
nan
inf
-inf
```

In [65]:

```
a = np.array([0, np.pi/3, np.e/2, np.pi, np.e])
print(np.sin(a))
print(np.cos(a))
print(np.tan(a))
```

```
[0.00000000e+00 8.66025404e-01 9.77684488e-01 1.22464680e-16
 4.10781291e-01]
[ 1.          0.5         0.21007866 -1.         -0.91173391]
[ 0.00000000e+00  1.73205081e+00  4.65389724e+00 -1.22464680e-16
 -4.50549534e-01]
```

In [78]:

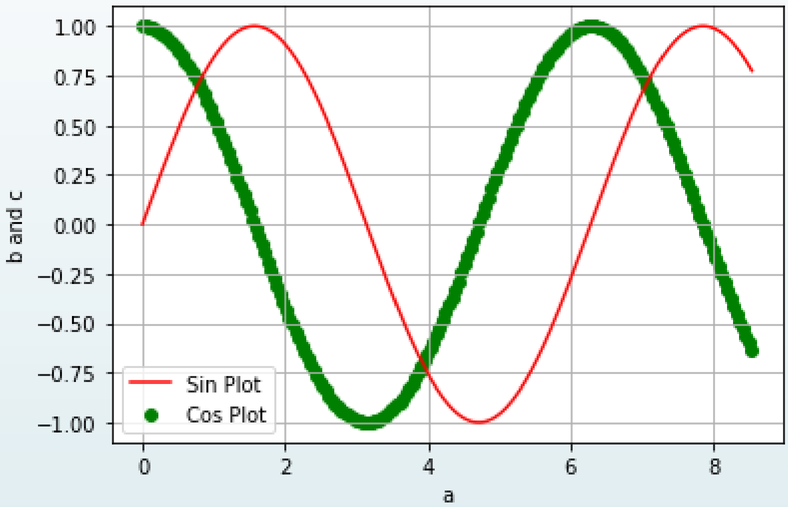
```
import matplotlib.pyplot as plt

a = np.linspace(0, np.e*np.pi, num=200)
b = np.sin(a)
c = np.cos(a)

plt.plot(a, b, color='red', label='Sin Plot')
plt.scatter(a, c, color='green', label='Cos Plot')

plt.xlabel('a')
plt.ylabel('b and c')

plt.legend()
plt.grid()
plt.show()
```



In [81]:

```
nlis = [[1, 2, 3], [11, 12, 13], [14, 15, 16]]
numpy_array = np.array(nlis)
print(numpy_array)
print(numpy_array.ndim)
print(numpy_array.shape)
print(numpy_array.size)
```

```
[[ 1  2  3]
 [11 12 13]
 [14 15 16]]
2
(3, 3)
9
```

In [93]:

```
#Accessing
print(numpy_array[0, 0])
print(numpy_array[0, 1])
print(numpy_array[0, 2])
print(numpy_array[1, 0])
print(numpy_array[1, 1])
print(numpy_array[1, 2])
print(numpy_array[2, 0])
print(numpy_array[2, 1])
print(numpy_array[2, 2])
print(numpy_array[0][0:3])
print(numpy_array[1][1:3])
print(numpy_array[2][0:2])
```

```
1
2
3
11
12
13
14
15
16
[1 2 3]
[12 13]
[14 15]
```